Thesis for The Degree of Licentiate of Engineering

Deterministic, Explainable and Resource-Efficient Stream Processing for Cyber-Physical Systems

Dimitrios Palyvos-Giannas

Division of Networks and Systems
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2019

Deterministic, Explainable and Resource-Efficient Stream Processing for Cyber-Physical Systems

Dimitrios Palyvos-Giannas

Technical Report No 200L
ISSN 1652-876X
Department of Computer Science & Engineering
Division of Networks and Systems
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2019.

“But how could you live
and have no story to tell?”
— Fyodor Dostoevsky

Abstract

We are undeniably living in the era of *big data*, where people and machines generate information at an unprecedented rate. While processing such data can provide immense value, it can prove especially challenging because of the data's *Volume*, *Variety* and *Velocity*. Velocity can be particularly important in environments that need to respond to incoming data in near real-time, such as cyber-physical systems. In such cases, the *batch processing* paradigm, which requires all data to be persistently stored and available, might not be appropriate. Instead, it can be desirable to perform *stream processing*, where unbounded datasets of streaming data are processed in an online manner, generating results quickly and thus significantly benefiting applications with strict latency requirements. However, it can be challenging for stream processing to provide the same guarantees and ease-of-use as traditional batch processing systems. This thesis studies ways to alleviate this by introducing techniques that make stream processing more predictable, explainable, and resource-efficient.

In the first part of the thesis, we study *determinism*, which can guarantee predictable and reproducible results in stream processing, regardless of the runtime system characteristics. We present *Viper*, a module for stream processing frameworks that provides determinism with a minimal performance impact. In the second part, we study *fine-grained data provenance*, which links each streaming result with the inputs that led to its generation. Fine-grained data provenance can help make stream processing easier to understand and debug. Additionally, it can reduce storage and transmission costs by allowing to maintain only the essential input data. We propose the *GeneaLog* framework that provides fine-grained data provenance in stream processing with minimal overhead. In the third part of the thesis, we explore *scheduling* and its use in stream processing for controlling resource allocation and achieving specific performance goals. We develop *Haren*, a framework that can be integrated into stream processing frameworks, providing custom thread scheduling capabilities. We study Haren's efficiency and its facilities that allow a user to control the resource allocation of a streaming system. We evaluate all three proposed frameworks with relevant streaming use cases from the real-world and illustrate their efficiency and ease-of-use.

Keywords

Stream Processing, Provenance, Determinism, Scheduling

Acknowledgment

Ralph Waldo Emerson writes that *"our chief want in life, is somebody who shall make us do what we can"*. Thus, I cannot begin this thesis without acknowledging my supervisor, Vincenzo Massimiliano Gulisano. I would like to thank him for his guidance, his strong belief in me and his insistence to always aim higher. Most of all, I would like to thank him for always having an open mind, for taking the time to listen and for being understanding when he really needed to. At the same time, I would like to express my genuine gratitude to my co-supervisor, Marina Papatriantafilou, for her patience, insightful suggestions, and her continuous assistance in all part of my studies.

Moreover, I would like to give special thanks to Yiannis, Ivan and Iosif for their invaluable advice about research, the Ph.D. experience and — why not — life in general. I remember reading somewhere that regular breaks increase productivity, so thank you, Hannah, Georgia, Christos, Valentin, Fazeleh, and Thomas for frequently dropping by my office to discuss everything and nothing. More than that, interactions with everyone in the division have helped me grow both personally and professionally, so I would like to thank all my close colleagues. Thank you, Adones, Ali, Aljoscha, Amir, Aras, Bastian, Bei, Beshr, Babis, Carlo, Elena, Elad, Joris, Karl, Katerina, Magnus, Nasser, Olaf, Oliver, Philippas, Romaric, Thomas P., Tomas, Valentin T., and Wissam. I would also like to thank all the administrative staff for making my day-to-day life easier, and especially the people I interact directly with: Eva, Marianne, and Rebecca.

I wish to acknowledge the financial support by the Swedish Research Council (Vetenskapsrådet), project "HARE: Self-deploying and Adaptive Data Streaming Analytics in Fog Architectures", with grant number 2016-03800.

Last, but by no means least, I would like to thank the people who made all this possible. My wonderful parents, who made countless sacrifices to ensure that I would be able to follow the path I wanted. My lovely sister, who was always there to ~~bother~~ support me. My fantastic friends, both in Sweden and in Greece, for making my free time much more enjoyable. And, of course, my incredible partner, Pinelopi for her love, support, and unlimited patience.

Dimitris Palyvos-Giannas
Göteborg, October 2019

List of Publications

Appended publications

This thesis is based on the following publications:

[A] I. Walulya, D. Palyvos-Giannas, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas "Viper: A Module for Communication-Layer Determinism and Scaling in Low-Latency Stream Processing"
Future Generation Computer Systems, vol. 88, pp. 297-308, 2018.

[B] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou "GeneaLog: Fine-Grained Data Streaming Provenance in Cyber-Physical Systems"
In submission to Journal Parallel Computing.

[C] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou "Haren: A Framework for Ad-Hoc Thread Scheduling Policies for Data Streaming Applications"
Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, 2019, pp. 19-30.

Other publications

The following publications were published during my PhD studies, or are currently in submission/under revision. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to the thesis.

[a] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou "GeneaLog: Fine-Grained Data Streaming Provenance at the Edge"
Proceedings of the 19th International Middleware Conference, Rennes, France, 2018, pp. 227–238.

Contents

List of Figures

1

Overview

1.1 Introduction

The late 1970s saw the introduction of the concept of a *database machine*, a dedicated computer to store and process data [2, 3]. A few years later, in 1984, the Teradata corporation surprised the market with the introduction of the DBC/1012 database machine, which could store and process up to a terabyte (10^{12} bytes) of information by taking advantage of multiple CPUs and hard disks [4]. Thirty years later, in 2014, the world would need 1 billion(!) of these machines to store all of its data. By 2018 this figure was 33 billion and it is expected to reach 175 billion (or 175 zettabytes) by 2025 [5].

It is undeniable that we are living in the era of *"Big Data"*. In 2017, *The Economist* wrote that *"The world's most valuable resource is no longer oil, but data"* [6]. A chilling confirmation of this statement came a few months later when the Cambridge Analytica scandal illustrated how one could take advantage of big data to swing the minds of voters in multiple elections [7]. Similarly to oil, which is not particularly useful before it is extracted from the ground, big data is not useful before it is processed. In this thesis, we focus on ways to process such data efficiently and quickly. However, before discussing processing, we need to clarify what big data is and how it can be useful.

The Identity of Big Data

Although there is no single definition of big data, it generally refers to *"the increase in the volume of data that are difficult to store, process and analyze through traditional database technologies"* [8]. Big data has four main characteristics, referred to as the four V's [9]: (i) *Volume*, which is at the petabyte scale, ruling out traditional databases and requiring new storage techniques, (ii) *Variety*, which refers to heterogeneous data points which can include text, audio, video, position reports etc, (iii) *Velocity*, which refers to the fast rate of arrival of such data and (iv) *Value*, which is contained within the massive datasets and can be extracted by combining the individual data points. Paraphrasing the above, big data is *"too big, too fast or too hard for traditional databases to process"* [10]. But where does this data come from?

One of the primary sources of big data is individuals and their online interactions. The World Wide Web and social networks such as Facebook and Twitter are significant contributors to the generation of big data. In 2019, more than 500 hours of video were uploaded to YouTube every minute [11]. Facebook's more than 2.32 billion users have shared more than 219 billion photos in a network of more than 140 billion friend connections [12, 13]. This data generation from individuals is expected to increase even further. It is estimated that, by 2025, the average person will have a digital data engagement over 4900 times per day, which translates to one interaction every 18 seconds [5].

Apart from online user interactions, another important source of big data are applications related to the Internet of Things (IoT). The IoT is about embedding numerous sensors in a multitude of everyday machines and devices, from smartphones and tablets to washing machines and refrigerators, bus stops and aircraft engines. These sensors collect vast amounts of heterogeneous data, for example geographical, logistical, health, and environmental. This data can provide great insights, enabling smarter services and more efficient processes. By 2030 there are expected to be one trillion IoT sensors, making them one of the most significant sources of big data. An essential characteristic of this data is that it is usually very noisy. In particular, there might be only very few critical events that need to be distinguished among thousands of data points, introducing further challenges in the analysis of such datasets [14].

The Value of Big Data

Regardless of where they originate, the analysis of big data can be extremely valuable in a myriad of applications. For example, big data was an essential contributor to the "boom" in Artificial Intelligence research that we are experiencing today [15]. Although many of the now-popular machine learning algorithms were already known, there was a lack of high-quality training data to make them useful. As companies like Google, Facebook, Microsoft, and Twitter started capturing and analyzing big datasets, it became possible to make revolutionary advancements in areas such as image and speech recognition, spam detection and text translation using machine learning techniques [16]. Nowadays, a large number of companies are taking advantage of big data to train the next generation of machine learning algorithms powering novel technologies such as self-driving cars.

Another example of the value of big data is the analysis of social network interactions, which can not only provide insights about behaviors and desires of individuals but also societies as a whole. Companies with access to vast amounts of user data, such as Google and Facebook, take advantage of this to correctly predict which advertisement will be most effective on an individual level. Apart from such use cases, social networks can potentially act as "social sensors", which can be utilized to create early warning systems by detecting unusual events (e.g., earthquakes) based on the real-time user behaviors [17]. Social network data can also be used in monitoring systems, giving live feedback about the behaviors and opinions of their users [14].

In the field of IoT and cyber-physical systems, data is not only useful for analysis purposes but also for making real-time decisions. In smart grids, for example, sensors (smart meters) in houses and businesses report electricity

consumption values in real-time. The operator can then query these reports to detect events such as blackouts or power surges [18–20]. Long-term power data can assist in planning future expansions or reductions of the grid. Furthermore, the cyber-physical system itself can take real-time corrective measures. For example, it can balance the power generation based on the current consumption, reducing redundancy and waste [14].

1.2 Processing Big Data

The real value of big data shines when we can extract insights from it through quick and efficient processing. Here, we discuss two prevalent processing models for big data, namely batch and stream processing, as well as the processing infrastructure that is required to support them.

1.2.1 Batch Processing

The MapReduce model was developed to process vast amounts of raw information such as crawled Web pages and transform them into useful representations, such as graph structures of Web documents [21]. MapReduce was innovative for its time because it could process datasets of huge *Volume* by combining hundreds or thousands of machines and parallelizing the computation. It significantly reduced the complexity of big data processing by hiding issues such as fault tolerance, load balancing, and data distribution and led the way for a myriad of big data processing frameworks such as Hadoop [22] and Spark [23]. Similarly to traditional database systems, these big data processing frameworks adopted the *batch* processing model. This model assumes that the data, although vast in size, is *bounded* and it is persistently stored somewhere, available for processing. Batch processing proved immensely successful, as it allowed processing data at the petabyte scale with very high throughput. However, there are limitations of batch processing that make it inadequate for an increasing number of modern use-cases. In this thesis, we focus on a different processing paradigm, called *stream processing*, which aims to address these limitations.

1.2.2 Stream Processing

The *Velocity* attribute of big data implies that, to maximize its value, such data needs to be analyzed immediately as it streams from its sources [24]. This can be crucial in cyber-physical systems such as industrial manufacturing machines which need to react to their environment in near real-time [5, 25]. The *batch processing* model, which includes relational databases and big data processing frameworks, is not capable of coping with such online processing requirements. In that model, the data is first stored and indexed and then queried by users in a polling fashion, leading to high latency results. These limitations illustrate the need for a different processing model, specially adapted to online data processing. This model is referred to as *stream processing*.

Stream processing frameworks are specialized tools for online data processing. They do not require all data to be persistently stored but, instead, can analyze *unbounded datasets* of streaming data in an online manner. In contrast with

the batch processing model, where data is queried only when the user explicitly asks for it, stream processing frameworks *continuously* process incoming data as it arrives [26]. This enables low latency results (down to milliseconds), giving the opportunity for faster insights and decision making.

Stream processing has been studied since the beginning of the 2000s and the academia proposed stream processing frameworks such as Aurora [27] and Borealis [28]. Nowadays, companies that deal with big data are also actively utilizing stream processing with their own frameworks. Notable examples include Google's DataFlow [29], Facebook's Puma, Stylus and Swift [30], Twitter's Storm and Heron [30, 31] and LinkedIn's Samza [32]. Some of these frameworks are published as open-source, available for anyone to use and study. Additionally, there is a multitude of other open-source streaming frameworks, the most noteworthy of which is Apache Flink [33].

1.2.3 Processing Infrastructure

Cloud computing is a technology tightly coupled with big data generation and processing. It is the realization of "computing as a utility" and it has arguably shaped the way people design and use applications. "The Cloud" has many definitions, including Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). These definitions cover high-level applications such as Gmail as well as much lower-level infrastructure such as hardware equipment that can be rented by the hour [8]. In this thesis, we consider cloud computing to be any hardware and software resources offered to the public in a pay-as-you-go model. This kind of computing was revolutionary when it first appeared because it removed the obligation to purchase and maintain expensive data-center infrastructure for purposes such as hosting web applications or processing massive amounts of data. Instead, the necessary infrastructure is provided by a cloud provider, and the cloud user pays only for the resources she consumes. More specifically, cloud computing has three main characteristics [34]: (i) the illusion of infinite computing resources, (ii) no up-front costs, and (iii) scaling on demand (and being charged accordingly). The third characteristic is especially important because it removes the costs associated with over or under-provisioning (i.e., having more or fewer resources than necessary) [35]. Related to this is the *"cost associativity"* of the cloud, which means that, in many cases, it costs the same to run one cloud instance for 1000 hours or 1000 instances for one hour. This makes cloud computing especially attractive for massively parallel data analytics applications. All of the above have made cloud computing a desirable paradigm for many big data analytics applications. However, as the volume and velocity of data increases at an unprecedented rate, it brings the cloud computing model to its limits, illustrating the need for hybrid, decentralized computational models [5].

Today, broad access to smartphones and various Internet-of-Things devices empowers individuals to generate more data than ever. Running streaming analytics on such data can be very valuable for extracting insights and also making real-time decisions [25]. However, as the number of connected devices grows rapidly every year, the established cloud-computing paradigm is reaching its limits. With a predicted 29 billion connected devices by 2022 [36], it is starting to become infeasible to transfer all raw data captured by such devices

to the cloud for processing. Even if the communication infrastructure follows the growth in the number of devices, much of the raw data captured by end-devices such as smartphones and IoT sensors is uninteresting, and thus it would be inefficient to utilize communication links and resources to transfer them to a remote cloud [37, 38]. Instead, it would be much more preferable to pre-process the data as close as possible to the sources and only transmit relevant information to the cloud for further processing.

Edge computing (also referred to as *fog computing* [39] or *cloudlets* [40]) tries to address this challenge by moving the processing closer to the end devices, in nodes such as routers, switches or base stations [25]. Thus, edge computing aims to increase the capabilities and responsibilities at the edges of the network, reducing the computational load at the *core*, which includes cloud data centers and the accompanying communication infrastructure. Because data processing in edge computing takes place in nodes at most a few hops away from the end device, it opens the possibility of much lower latency results. These low-latency capabilities can be especially important in cyber-physical systems which need to respond in real-time to changes in the environment and cannot afford to wait for the duration of a round-trip communication to a remote cloud [25, 37].

In stream processing, in particular, the edge layer can be used to provide a hierarchical distribution of the computation. It can facilitate analytics strategies where edge nodes pre-process, pre-filter, aggregate, and encode data generated in end-devices before transmitting it to the cloud [41–43]. Such processing can remove a lot of the noise in the input (i.e., large amounts of unimportant events) and possibly provide stronger privacy guarantees by not transmitting sensitive information to remote cloud servers [38, 40, 44, 45]. Moreover, in certain applications, computations that are unsuitable for end-devices (e.g., smartphones) due to middleware or hardware limitations can be offloaded to edge nodes instead. This provides all the advantages mentioned above together with a reduction of the energy consumption of battery-powered end-devices [25]. In contrast to the cloud, edge nodes communicate only with end-devices in their proximity and are thus *location-aware*. This opens new exciting opportunities for smart aggregation and privacy enforcing features [40, 46, 47]. For example, a camera in a connected vehicle that detects an accident can notify the nearest base station which in turn may notify the drivers of nearby cars.

Along with the opportunities, edge architectures present new research and engineering challenges. The nodes deployed in such architectures are often heterogeneous and resource-constrained. Thus, it is essential to take this heterogeneity into account, providing appropriate programming models with support for both task and data-level parallelism, able to take advantage of multiple processing units [48, 49]. Additionally, the hardware constraints at the edge demand lightweight software and algorithms that do not overwhelm the available resources [25, 39].

1.3 Aspects of Stream Processing

In this section, we discuss various aspects of stream processing that are studied in the thesis. The stream processing paradigm deals with processing streams

of unbounded data. This is done by processing frameworks called *Stream Processing Engines (SPEs)*. An SPE runs one or multiple streaming queries. A streaming *query* is a directed graph of operators connected through streams. A *stream* is a (potentially infinite) sequence of objects, possibly sharing the same schema. An *operator* is the basic processing unit in stream processing, ingesting one or more streams, processing the data items in an online manner and producing one or more streams as output. Special *Ingress* operators (also called *Sources* or *Spouts* [33, 50]) deliver the streams to the streaming query. The final results are handled by *Egress* operators (also called *Sinks* [33]).

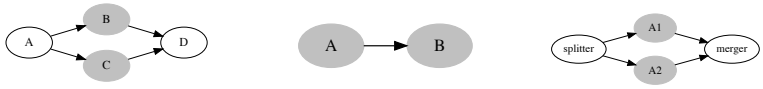
Streaming systems aim to process unbounded data streams in an *online* manner. For this reason, their main *Quality-of-Service (QoS)* goals are low latency and high throughput [51]. *Latency* in stream processing generally refers to the amount of time it takes until an input object produces a result. *Throughput* refers to the rate of streaming data that can be ingested by the streaming query per unit of time. Streaming systems might also set additional, extra Quality-of-Service goals such as memory consumption, balanced load or utilization, depending on application-specific requirements [52, 53].

1.3.1 Parallel Processing

Modern streaming systems take advantage of multi-core and distributed architectures to cope with the increasing data volume and velocity, by *parallelizing* their computations whenever possible. Parallelization refers to splitting the computation and assigning different tasks to separate computational units. In streaming, there are two main methods of parallelization: task parallelization and data parallelization [51]. *Task parallelization* refers to running different, independent operators in parallel whereas *data parallelization* (or *operator fission* [54]) is about running multiple identical instances of a single operator on different subsets of the data in parallel. In both cases, the parallel instances run independently, either on different CPU cores or even on different machines.

Figure 1.1 shows an overview of the different kinds of parallelism. In task parallelism, independent operators run in parallel (`B, C` in the figure). *Pipeline parallelism* is a subset of task parallelism, where the parallel operators have a producer-consumer relationship (`A, B` in the figure). In data parallelism, a *splitter* partitions the input data, which is then processed by multiple identical parallel operator instances (`A1, A2` in the figure). The splitter's strategy defines how the data is distributed to the different parallel instances (see [51] for an overview of splitting strategies). Finally, the parallel input streams are merged into a single stream by a *merger*, which can also enforce in-order delivery of the tuples to the downstream operator. Both the splitter and the merger can be either distinct operators or integrated into the processing operator logic.

Data parallelization can be especially useful if the velocity of the data arriving at an operator (i.e., its arrival rate) exceeds that operator's maximum processing speed. In such a case, the Quality-of-Service of the whole streaming pipeline might start to degrade. More specifically, since the operator will not be able to process data quickly enough, its input queue(s) will start growing in size, leading to an increase in the latency of its output tuples. This latency increase will cascade downstream, potentially increasing the overall latency of the streaming pipeline. Moreover, if the system uses a *backpressure* [55]



(a) Task parallelism. **(b)** Pipeline parallelism. **(c)** Data parallelism.

Figure 1.1: Different types of parallelism.

mechanism, the upstream operators will slow down to match the speed of the bottleneck operator, thus decreasing the throughput of the system. Using data parallelization, the operator's input stream is partitioned and, instead of one operator instance processing the whole input, multiple identical instances of the operator process different parts of it. Since each instance processes less data individually, enough parallel instances can potentially eliminate the performance bottleneck, improving the Quality-of-Service of the streaming pipeline. Special care needs to be taken when parallelizing stateful operators so that they end up with a self-contained state.

Applications that rely on parallelism require a *synchronization* mechanism, so that parallel threads access and alter shared state safely [56]. The simplest form of this is *coarse-grained synchronization*, where locks are used to synchronize large parts of the functionality. Applications such as stream processing, which have strict throughput and latency requirements, require more advanced, fine-grained synchronization. *Non-blocking synchronization* methods are of particular interest since they do not rely on locks but take advantage of build-in atomic operations of the system (e.g., Compare-And-Swap). Such methods can have the desirable characteristic of *lock-freedom*, guaranteeing that at least one of the parallel thread will always be making progress.

1.3.2 Predictable and Explainable Stream Processing

The increasing demand for stream processing from academia, industry, and individuals means that more and more analysts - many of which are not experts in stream processing - are required to write streaming queries. For an analyst, the transition from a traditional database to a streaming system is not always straightforward because of the inherently temporal nature of streaming. Even simple database queries such as "compute the average electricity consumption per city" become more complicated in the streaming context because the analyst has to consider timing issues such as windowing, event, and processing times and delayed or out-of-order data [29]. This can make it difficult for non-experts to understand the results of streaming programs and especially challenging to write and debug streaming queries [57]. Additionally, prototyping and testing streaming applications can also be more difficult because the data is flowing and thus it might not always be possible to run a new version of the query on the same data to verify correctness [30].

The above trends indicate a need for streaming systems that are *explainable* and present users with easily understandable results. Additionally, streaming systems need to be *predictable* and offer some degree of reproducibility [58]. There are attempts to mitigate these issues by bringing streaming systems

closer to a relational model where streams and tables are treated as "two sides of the same coin" [59]. Here, we discuss two orthogonal approaches that can increase understanding and predictability in streaming pipelines: *provenance* and *determinism*.

Explaining and Filtering Data Through Provenance

The proliferation of stream processing, together with the growing size and complexity of streaming pipelines makes it very desirable to be able to explain their results. For example, social networks such as Twitter or Facebook heavily rely on streaming pipelines to quickly detect events when their users start to talk about them, acting as social sensors [17,30]. These events can have a significant societal and political impact, making it critical to assess their correctness and trustworthiness [60]. Similar requirements exist for streaming systems deployed in mission-critical scenarios (e.g., detecting fires) in cyber-physical systems.

One technique for achieving this is *provenance*. In general, provenance is "any information that describes the production process of an end product" [61]. In streaming systems, in particular, we are usually interested in *fine-grained data provenance*, which traces every output event to the input events that contributed to it. A streaming system that provides fine-grained data provenance can explain precisely how its results came to be, easing debugging and making the system more explainable. Moreover, provenance allows the system to store the source data that led to the generation of important events, enabling reproducibility and reducing data storage and transmission requirements.

Fine-grained data provenance can be especially crucial in cyber-physical systems such as sensor networks [60]. One use of provenance in such systems is to explain critical events and allow further investigation. A relevant example could be a smart grid system which uses a streaming pipeline to produce alerts for blackouts if multiple homes report zero consumption during the same time window. In case of such an alert, the human operator would need to quickly find out which smart meters from which houses reported loss of power, in order to take the necessary steps to alleviate the issue. This goal could be easily achieved with fine-grained data provenance, which would be able to report the exact measurements that generated the blackout alert.

Another issue with modern cyber-physical systems is that they generate vast amounts of data. A modern vehicle senses dozens of gigabytes per day [62], jet engines can generate a terabyte of data per 24-hour period [63] and autonomous cars are expected to produce over three terabytes of data per hour [5]. Depending on the application, it can be costly or even infeasible to store or transmit all this data to the cloud for processing. One alternative is to process the data locally (or delegate the processing to edge nodes) and use provenance to store and transmit only noteworthy inputs (i.e., those which led to the generation of critical events), removing unwanted "noise" from the data [38].

Achieving Predictability through Determinism

Apart from being explainable, being predictable can also be very beneficial for a streaming system by making it easier to understand and debug streaming queries. One way to achieve predictability is to guarantee *deterministic* processing. In

a nutshell, deterministic processing implies that, for a specific input, the streaming system always produces the same output.

In stream processing, data is pushed through a directed graph of operators. Depending on the execution model, some or all of these operators can execute in parallel. Similarly to traditional parallel programs, non-deterministic behavior can arise merely due to the parallel processing and the data interleavings during runtime. This can happen even if there is nothing inherently non-deterministic in the program logic itself [64]. For example, an operator might be receiving data from multiple parallel streams, either from parallel instances of another operator or from different operators. If the system does not explicitly enforce determinism, the interleavings of the input tuples to that operator might differ between executions, depending on runtime characteristics (e.g., scheduling decisions, processing load). Consequently, the operator might produce different outputs for the same input, exhibiting non-deterministic behavior. Such behavior can complicate debugging and make it more difficult to reason about streaming programs.

We mainly focus on *external-determinism* or *determinacy*, where successive program executions with the same input always produce the same final output, regardless of the runtime characteristics of the system. A streaming system can guarantee deterministic processing if two requirements are met. First, determinacy needs to be satisfied for every operator, meaning that, for the same input sequence, each operator will produce the same output sequence. Second, determinacy needs to be satisfied for the data flows between the operators so that the sequences of objects in these input streams is always the same, regardless of runtime characteristics such as the parallelism degree. As discussed later in the thesis, the trade-offs of determinism are degraded performance and programmability [64]. In this thesis, we study ways to minimize such drawbacks by enforcing determinism more efficiently.

1.3.3 Scheduling

As stream processing pipelines move closer to the edge and run in possibly resource-constrained devices, it becomes more and more important to efficiently utilize all available computational resources. At the same time, both edge and cloud processing nodes can run heterogeneous applications with vastly different priorities and Quality-of-Services requirements, making it critical to prioritize specific computations compared to others. These aspects indicate the need for a way to control resource allocation in data processing. Such control can be achieved with careful *scheduling*.

In general, scheduling refers to the process of assigning specific units of work to specific resources. In stream processing, in particular, we identify two main types of scheduling: resource scheduling and thread scheduling. *Resource scheduling* or *operator placement* deals with picking where each processing unit (i.e., streaming operator) is deployed and can be used to provide better load balancing and reduce communication cost, which in turn can improve other Quality-of-Service metrics such as latency and throughput [53, 65]. *Thread scheduling* is orthogonal to the resource scheduling and much more fine-grained, choosing which computation to prioritize inside each processing group [52]. This

thesis focuses on utilizing thread scheduling to achieve specific performance goals and satisfy the Quality-of-Service goals of heterogeneous streaming queries.

1.4 Research Problems

1.4.1 Parallelism and Determinism

As the volume and velocity of streaming data increases, data parallelization becomes critical for a streaming system to achieve the desired Quality-of-Service [51,66–68]. However, parallelization comes with its own set of conflicting goals. On one hand, parallelization performs best when there is minimal synchronization, and the parallel instances of the operators work independently as much as possible. On the other hand, correctness in the form of determinism increases the synchronization requirements of the system [64, 69, 70]. This is because, to ensure determinism, not only do operator implementations need to be deterministic, but there also needs to be a consistent ordering of their input tuples. The ordering requirement is especially important when such operators are fed by multiple parallel input streams. To ensure determinism in such cases, the tuples from all input streams need to be sorted in timestamp order before they can be processed by the next operator. In some SPEs, sorting is done at the *Operator-Level* by dedicated merge-sorting operators [71, 72]. However, such operators can negatively affect performance, defeating the purpose of data parallelization. Additionally, the need for adding such operators to the queries conflicts with the desire for *syntactic transparency* which would allow a query to be parallelized with minimal or no configuration from the user [73]. Thus, our first research question is *"Can we achieve highly-parallel and deterministic processing in streaming transparently, while minimizing the performance impact of determinism?"*

1.4.2 Fine-Grained Data Provenance in Streaming

As discussed in § 1.3.2, fine-grained data provenance can be beneficial in streaming systems, allowing the user to verify correctness, easing debugging and maintaining relevant source data related to critical events [57, 61, 74]. Fine-grained data provenance in streaming needs to link every output tuple back to the source tuples that contributed to it. However, due to the potentially high volume and velocity of the data, provenance is an intrinsically heavy operation. This implies that the performance of a streaming application can be limited by the efficiency of its provenance capture [60]. State-of-the-art approaches for streaming provenance use instrumented operators that enrich tuples with provenance specific annotations that allow the linking of output to source tuples to occur [60]. However, these annotations can grow arbitrarily large, putting pressure on the system's memory. Additionally, these methods require the maintenance of all source data for some time, which can further add to the memory and processing requirements. Because of the above, these state-of-the-art approaches can be prohibitive in applications maintaining large states or applications that need to run in resource-constrained devices, such as those deployed at the edge of cyber-physical systems [25, 39, 75]. Thus, our

second research question is *"Can we maintain fine-grained data provenance in streaming systems with minimal processing and memory overhead?"*

1.4.3 Customizable Thread Scheduling

Careful scheduling can prove particularly useful in modern SPEs, enabling better utilization of system resources and improving the Quality-of-Service [27, 53, 65, 76]. Thread scheduling, in particular, gives fine-grained control for prioritizing specific computations and achieving precise Quality-of-Service goals [52]. Over the years, there has been extensive research in thread scheduling, leading to many techniques for optimizing particular performance metrics [27, 77–81]. However, custom thread scheduling techniques for streaming have mostly remained research prototypes. Widely adopted SPEs such as Apache Flink [33] or Apache Storm [50] offer no option for custom thread scheduling. Instead, they rely on the operating system to schedule their processing threads. This is because thread scheduling involves many low-level details and is difficult to implement efficiently and correctly. Additionally, scheduler implementations are usually bound to a specific SPE, making it difficult to extend or port to other SPEs. Unfortunately, relying on the operating system for scheduling is not always ideal. The operating system lacks *application-awareness*, i.e., it is unaware of the various Quality-of-Service goals of the streaming queries, potentially leading to sub-optimal performance. These issues give rise to our third research question: *Can we provide resource-efficient and application-aware scheduling for streaming systems?*

1.5 Thesis Contributions

1.5.1 Communication-Layer Determinism

We begin this work in Chapter 2 by tackling the first research question regarding highly parallel stream processing with determinism guarantees, which was introduced in § 1.4.1. We propose a modular and transparent method to guarantee determinism in parallel stream processing without sacrificing efficiency. To achieve this, we replace the traditional dedicated merge-sort operators with a new technique. In particular, we build upon the *ScaleGate* [82] data structure which, in contrast with previous approaches, relies on lock-free synchronization to merge-sort the parallel streams. Moreover, ScaleGate allows us to move the enforcement of determinism from the Operator (Application) Layer to the Communication (Middleware) Layer, providing the opportunity for much higher performance. We distill our efforts in the implementation of the *Viper* module, which can be integrated into SPEs, providing determinism facilities transparently to the user with a minimal performance impact. We evaluate this technique in realistic scenarios from vehicular networks and smart grid systems and observe significant benefits in the Quality-of-Service (throughput and latency) as well as in the energy efficiency of the system.

1.5.2 Low-Overhead Streaming Provenance

In the second part of the thesis, in Chapter 3, we answer the second research question and present GeneaLog, a new technique for fine-grained data provenance in deterministic streaming applications. GeneaLog advances the state-of-the-art by not requiring variable-length annotations for provenance capture. Instead, it uses small, fixed-size annotations, reducing the memory overhead of provenance. Additionally, it removes the requirement for temporary storage or transmission of all the source data by taking advantage of the memory management of the process to distinguish source tuples that are actually important for provenance. Lastly, it gives the option to distribute the provenance computation to separate nodes. We evaluate a fully implemented prototype of GeneaLog in two different SPEs using real-world queries and study its performance and scalability characteristics, observing considerable improvements (sometimes more than an order of magnitude) compared to the state-of-the-art.

1.5.3 Custom Scheduling for Streaming Systems

In Chapter 4, the last part of the thesis, we answer the third research question by exploring the scheduling problem in stream processing. In particular, we focus on *thread-scheduling*, i.e., choosing the allocation of the CPU threads to the streaming operators in order to meet specific performance goals. Instead of dedicating a thread for every streaming operator and delegating the scheduling task to the operating system, we study application-level thread scheduling, where the streaming system is responsible for its scheduling decisions. We distill the abstractions needed to quickly implement diverse scheduling policies in an SPE as well as the primitives that the SPE needs to support for this to occur. Moreover, we use our observations to design and implement `Haren`, a general scheduler for streaming systems. `Haren` can be integrated into a streaming system and offer custom thread scheduling facilities with minimal effort from the user. We thoroughly evaluate `Haren` with different scheduling policies using hardware that can be deployed at the edge of cyber-physical systems, where correct scheduling decisions can be of paramount importance [81]. We observe that `Haren` manages to meet the goals of a variety of scheduling policies and, in many cases, outperforms the dedicated threads approach.

1.6 Conclusions and Future Work

This thesis studies several optimizations for stream processing with a focus on cyber-physical systems. In this age of big data of high volume, velocity, and variety, we present techniques to make stream processing systems deterministic and explainable without sacrificing resource-efficiency. We propose and evaluate three fully implemented prototypes, Viper, GeneaLog, and Haren, which advance the state-of-the-art in determinism, provenance, and scheduling in stream processing, respectively. In particular, Viper reduces the communication and synchronization costs of parallel operator instances within an SPE and boosts the scale-up potential of streaming queries, being able to increase throughput up to 70% and reduce the energy consumption by half. GeneaLog provides

fine-grained data provenance in streaming with minimal overhead in both intra and inter-node deployments, greatly outperforming the state-of-the-art. Lastly, Haren easily integrates into an SPE and allows the implementation of user-defined thread scheduling policies with minimal effort, enforcing these policies efficiently by parallelizing the work and outperforming other, widely used approaches.

In future work, it would be interesting to explore further how to take advantage of the heterogeneous architectures present at the edge of cyber-physical systems, using GPUs and other accelerators to improve stream processing. Additionally, elasticity at the edge is an important direction that needs to be explored further, allowing to add or remove processing power in the form of edge nodes depending on the processing requirements. Finally, location-aware processing is an exciting direction, along with better visualization techniques of stream processing that would further aid in understanding and debugging streaming pipelines.